IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

Reconcilable and Undoable File System

Inventor(s):
Marc Shapiro
James William O'Brien
Nuno Preguiça

ATTORNEY'S DOCKET NO. MS1-1509US
CLIENT'S DOCKET NO. 303448.1

RECONCILABLE AND UNDOABLE FILE SYSTEM

TECHNICAL FIELD

The described subject matter relates to file systems. More particularly, the subject matter relates to reconciling one file system state with another file system state.

BACKGROUND

Networked and mobile computers enable different users to share files and directories of information, and enable a single user to utilize a file system across multiple computers. The files and directories are typically managed on the computers by a file management system that applies file management rules, such as naming conventions and file and directory organizational rules. A file system may be replicated on multiple computers and independently modified on those computers with various file system commands. However, when the replicated file system is independently modified on multiple computers, the states of file systems on the computers may diverge.

Therefore, after replicated file systems are independently modified, users attempt to synchronize the divergent file system states; i.e., reconcile the divergent file system states into a single, common file system state. Nevertheless, during synchronization, conflicts may arise between the divergent file system states. An exemplary conflict includes one or more file system state conditions that violate the rules of the file management system. Such conflicts may result in an inability

to fully synchronize the divergent file system states, at least without manual intervention.

Most existing file management systems identify conflicts using only the file system states as those states exist at the time of synchronization. With such limited information, existing file management systems have not offered flexible approaches for resolving such conflicts. Some file management systems, for example, simply disallow certain actions that violate predetermined file management system rules regarding file names and other properties, often resulting in a partially reconciled file system. To partly address this situation, some file management systems may also allow manual reconciliation, which may be difficult or impossible with the partially reconciled file system.

For example, two different files having the same name in the same directory in different file system states can violate certain file management system rules. However, in this situation, an existing file management system simply does not synchronize the conflicting conditions, while synchronizing non-conflicting conditions that may have been dependent upon the conflicting condition (e.g., not copying conflicting files into a directory tree, while copying all other files into the directory tree). The result is often a single, common file system state that does not match either original file system state and is not easily reconcilable through manual intervention.

In addition, because most existing file management systems choose between conflicting file system states, these file management systems do not allow a user to reverse particular commands that led up to a file system state. After entering a file system command, a user might want to eliminate the command to avoid a conflict. A user might also want to retroactively eliminate a command because the command was entered by mistake, or for any other reason. If, for example, one user deletes a directory with all of the directory's contents and another user edits a document in the directory, the resulting states would conflict because of these two user interactions; however, existing systems do not allow for a user to selectively eliminate one of the commands in order to remove the conflict or reverse a mistakenly entered command.

SUMMARY

Implementations described and claimed herein solve the discussed problems, and other problems, by providing flexible approaches to reconcile divergent file system states and/or undo file system commands that may lead to the divergent file system states. Exemplary systems maintain records of file system commands that cause changes to file system states and enable users to selectively undo individual file system commands (or command sets) and/or choose proposed non-conflicting file system schedules.

Exemplary methods, systems, and devices have been developed for receiving a file system command to update a file system state, decomposing the file system command into one or more primitive actions representative of the file system command, and generating one or more log constraints associated with the one or more primitive actions.

An exemplary system includes a reconcilable file system operable to receive file system commands and generate a set of corresponding primitive actions. The exemplary system may further include a reconciliation engine operable to receive the set of primitive actions and reconcile the set of primitive actions with another set of primitive actions to generate a reconciled file system state.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates two exemplary reconcilable and undoable file systems in communication with a reconciliation engine.

Fig. 2 illustrates another implementation of two exemplary reconcilable and undoable file systems in communication with a reconciliation engine.

Fig. 3 illustrates an exemplary reconcilable and undoable file system that may be implemented in the systems of Figs. 1-2 for receiving file system

commands and generating primitive actions and log constraints corresponding to the file system commands.

Fig. 4 illustrates an exemplary reconciliation engine receiving two exemplary action logs and generating a non-conflicting action schedule.

Fig. 5 illustrates another exemplary reconciliation engine receiving two other exemplary action logs and proposing two exemplary non-conflicting action schedules.

Fig. 6 illustrates an exemplary reconciliation operation for reconciling file system states using primitive actions.

Fig. 7 illustrates an exemplary decomposing operation for generating one or more primitive actions representing a file system command.

Fig. 8 illustrates an exemplary reconciliation operation for collecting object constraints, generating a non-conflicting schedule, and committing the non-conflicting schedule.

Fig. 9 illustrates a reconciliation operation for reconciling one or more logs of primitive actions.

Fig. 10 illustrates a graphical user interface (GUI) for enabling a selective undo operation to selectively undo unwanted commands.

Fig. 11 illustrates another graphical user interface (GUI) for enabling a user to selectively undo unwanted commands.

Fig. 12 illustrates an exemplary system that provides a suitable operating environment to reconcile and/or undo file system operations.

DETAILED DESCRIPTION

Exemplary methods, systems, and devices are disclosed for reconciling conflicting file system states and/or selectively undoing unwanted file system commands. Generally, file system commands are analyzed to identify conflicts among two or more file system states. More specifically, the file system commands are translated into primitive actions and logged. Log constraints and object constraints representing relationships between primitive actions are used to identify conflicting primitive actions.

An Exemplary System for Reconciling File System States and/or Undoing File System Commands

Fig. 1 illustrates a reconcilable and undoable file system configuration 100 having two exemplary reconcilable and undoable file systems in communication with a reconciliation engine. A Personal Digital Assistant (PDA) 102 employs a reconcilable file system (RFS) 104 that facilitates reconciling a file system state on the PDA 102 with a file system state on a desktop computer 106. The desktop computer 106 employs a RFS 108 also facilitating reconciliation of the file system states of the PDA 102 and the desktop computer 106.

The PDA 102 is in operable communication with a desktop computer 106, whereby a file system state of the PDA 102 may be reconciled with a file system state of the desktop computer 106. The desktop computer 106 includes a reconciliation engine 112 for reconciling the file system states of the PDA 102 and the desktop computer 106. The PDA 102 employs a reconciliation interface 114 for communicating and receiving file system state data to and from the reconciliation engine 112. Similarly, the desktop computer 106 employs a reconciliation interface 116 for communicating and receiving file system state data to and from the reconciliation engine 112. File system state data from both of the PDA 102 and the desktop computer 106 are received by the reconciliation engine 112 to be reconciled.

A first log 118 has file system state information about the state of the file system at the PDA 102. Similarly, a second log 120 has file system information about the state of the file system at the desktop computer 106. The log 118 receives the file system state information associated with the PDA 102 from the RFS 104. The log 120 receives the file system state information associated with the desktop computer 106 from the RFS 108. The reconciliation interface 114 accesses the log 118 and communicates the log information to the reconciliation engine 112. The reconciliation interface 116 accesses the log 120 and communicates the log information to the reconciliation engine 112.

The reconciliation engine 112 collects the log information and generates reconciling file system state information for reconciling the file system states of the PDA 102 and the desktop computer 106 to a common file system state. The reconciliation engine 112 may communicate the reconciling file system state information to the RFS 104 and the RFS 108. The RFS 104 uses the reconciling file system state information to update the file system state of the PDA 102 to the common file system state. Likewise, the RFS 108 uses the reconciling file system state information to update the file system state of the desktop computer 106 to the common file system state.

The PDA RFS 104 and the desktop computer RFS 108 receive file system commands from applications or users that cause changes in the file system states of the PDA 102 and the desktop computer 106, respectively. The file system commands received by the RFS 104 may differ from the file system commands received by the RFS 108. For example, the file system commands received by the PDA RFS 104 may move a file from one directory to another, while file system commands of the desktop computer RFS 108 may cause a directory to be deleted. If the file system commands received by the PDA 102 differ from the file system commands received by the desktop computer 106, the resultant file system states of the PDA 102 and the desktop computer 106 may diverge.

The PDA RFS 104 and the desktop computer RFS 108 translate the file system commands into primitive actions representative of the file system

commands. Primitive actions are basic operations performed to carry out a file system command. More than one primitive action may be required to carry out a file system command. The PDA RFS 104 and the desktop RFS 108 also generate log constraints describing relationships between primitive actions. For example, a log constraint may indicate that two primitive actions must be performed in a particular order. Exemplary primitive actions and log constraints are discussed in more detail with regard to Fig. 3.

The RFS 104 stores primitive actions and log constraints in the log 118. The RFS 108 stores primitive actions and log constraints in the log 120. As will be appreciated, the primitive actions represent component actions of file system commands, which influence the file system states of the PDA 102 and desktop computer 106. Therefore, the primitive actions represent updates or modifications to the respective file system states. As such, the log 118 holds updates to the file system state of PDA 102, and the log 120 holds the update to the file system state of the desktop computer 106.

As discussed in more detail herein, the updates to the file system states of the PDA 102 and the desktop computer 106 are tentative updates, in that they may be modified or canceled later by reconciling information from the reconciliation engine 112. The reconciliation engine 112 processes the tentative updates (i.e., the primitive actions and the log constraints) from the PDA 102 and the desktop computer to attempt to reconcile the divergent file system states.

The reconciliation interface 114 communicates the tentative updates (i.e., the primitive actions and the log constraints) from the log 118 to the reconciliation engine 112. The reconciliation interface 114 may communicate the tentative updates in a batch mode, incrementally, or otherwise. Likewise, the reconciliation interface 116 communicates the tentative updates from the log 120 to the reconciliation engine 112 as a batch, incrementally, or otherwise. In one implementation, the reconciliation engine 112 requests the tentative updates from the reconciliation interfaces 114 and 116 periodically. In another implementation, the tentative updates are communicated to the reconciliation engine 112 as they are generated during real-time. The reconciliation interfaces 114 and 116 may perform data formatting operations, such as data encoding, in order to prepare the log information for communication to the reconciliation engine 112.

In one implementation, the reconciliation engine 112 analyzes the tentative updates from the PDA RFS 104 and the desktop computer RFS 108, to attempt to resolve conflicts, if any, between the file system states of the PDA 102 and the desktop computer 106. The reconciliation engine 112 also receives object constraints from the desktop computer RFS 108 via the reconciliation interface 116. An object constraint is a relationship between a primitive action from one computing device, such as the PDA 102, and a primitive action from another computing device, such as the desktop computer 106. Object constraints and

methods of communicating and analyzing object constraints are discussed in further detail below with reference to Fig. 8.

Using the primitive actions, the log constraints, and the object constraints, the reconciliation engine 112 generates a schedule of non-conflicting primitive actions. The non-conflicting primitive actions are selected such that when they are executed on the PDA 102 and the desktop computer 106, the file system states of the PDA 102 and the desktop computer 106 will be identical. The schedule of non-conflicting primitive actions is communicated to the PDA RFS 104 and the desktop computer RFS 108. Subsequently, the schedule of non-conflicting primitive actions can be committed to the file systems of the PDA 102 and the desktop computer 106, using methods discussed herein.

After receiving the schedule of non-conflicting primitive actions, the RFS 104 starts from a previous committed file system state and executes or "replays" the non-conflicting actions to arrive at a new committed file system state at the PDA 102. Similarly, the RFS 108 starts from a previous committed file system state and executes the non-conflicting actions to arrive at the same new committed file system state at the desktop computer 106. After the non-conflicting actions from the reconciliation engine 112 are played on the PDA 102 and the desktop computer 106, the PDA 102 and the desktop computer 106 have a replica of a reconciled file system state.

Fig. 2 illustrates an alternative configuration 200 for reconcilable files systems. In this particular configuration, a PDA 202 and a desktop computer 206 are in communication with a third computing device, such as a server computer 208. The server computer 208 has a reconciliation engine 212 in communication with a reconcilable file system (RFS) proxy 213 for performing reconciliation operations similar to the reconciliation operations described above with respect to the reconciliation engine 112 of Fig. 1.

The PDA 202 includes a RFS 216, a log 218, and a reconciliation interface 220 operable to perform reconciliation operations analogous to the reconciliation operations of the RFS 104, the log 118, and the reconciliation interface 114, discussed above with respect to Fig. 1. The desktop computer 206 includes a RFS 222, a log 224, and a reconciliation interface 226 operable to perform reconciliation operations analogous to the reconciliation operations of the RFS 108, the log 120, and the reconciliation interface 116, discussed above with respect to Fig. 1.

The RFS proxy 213 interfaces between the client devices (i.e., the PDA 202 and the desktop computer 206) and the reconciliation engine 212. One implementation of the RFS proxy 213 provides the reconciliation engine 212 with object constraints that represent a relationship between primitive actions from the two client devices. For example, an object constraint may specify that an object (e.g., a file) from the desktop computer 206 cannot have the same name as another

object (e.g., a directory) from the PDA 202. The RFS proxy 213 may have data that is specific to the file systems, commands, or actions employed by the client devices that enable the proxy 213 to generate the object constraints. The proxy 213 may also be programmed to take into account application-specific semantics when providing object constraints. Exemplary object constraints and how they relate to application-specific semantics are discussed in further detail below.

While Fig. 2 illustrates a reconciliation engine 212 operating on a third computing device, such as a server computer 210, in another implementation, the reconciliation engine 212 may reside and operate on one or both of the PDA 202 or the desktop computer 206. Also, the server computer 210 may include an RFS, logs, and one or more reconciliation interface(s) in operable communication with the reconciliation engine 212.

Fig. 3 illustrates an exemplary reconcilable file system (RFS) 302 to facilitate reconciling and/or undoing file system commands that result in divergent file system states. More specifically, the RFS 302 is operable to receive file system commands and generate primitive actions and log constraints corresponding to the file system commands.

The RFS 302 is in operable communication with an input/output (I/O) module 304, which sends file system commands and user-provided and application-provided log constraints to the RFS 304. The I/O module 304 interfaces between applications, user input devices and the RFS 302. The I/O

module 304 may receive file system commands to edit, delete, move, or otherwise change the characteristics of the files and directories. By way of example, one file system command is 'mkdir', which creates a directory.

The I/O module 304 may also receive log constraints provided by the user or an application. A log constraint represents a relationship between two primitive actions. For instance, a log constraint might indicate that a file delete command and a separate file creation command are to be executed together. The I/O module 404 recognizes file system commands, user-provided log constraints, and application-provided log constraints, and directs the commands and constraints to the RFS 302.

The RFS 302 translates the received file system commands into one or more primitive actions and stores the primitive actions in a log 306. In one particular implementation of the RFS 302, the RFS includes a decomposition module 303 and a recording module 305. The decomposition module 303 receives the file system commands and decomposes the file system commands into one or more primitive actions. The decomposition module 303 may also generate additional log constraints representing a relationship between two primitive actions. The decomposition module 303 communicates the primitive actions and the log constraints to the recording module 305. The recording module 305 records the primitive actions and the log constraints in the log 306.

In one embodiment of the RFS 302, the decomposition module 303 accesses a look-up table 308 to map file system commands to associated primitive actions. The look-up table 308 is a directory of file system commands, as shown in a 'Command' list 310, primitive actions, as shown in an 'Actions' list 312, and log constraints, as shown in a 'Constraints' list 314. In this implementation, the decomposition module 303 can match a received file system command to one of the file system commands in the Command list 310, and read corresponding primitive actions from the Actions list 312, and corresponding log constraints from the Constraints list 314.

To illustrate, assume a 'del dir' (i.e., "delete directory and directory contents") file system command is received by the RFS 302. The decomposition module 303 searches for and finds the 'del dir' command in the Command list 310. The decomposition module 303 then finds a corresponding set of primitive actions in the Actions list 312. In this example, the file system command 'del dir' correspond to three kinds of primitive actions: 'del files,' 'del subdir,' and 'del dir.' These three kinds of primitive actions indicate that in order to delete a directory in the file system, first all the files of the directory and subdirectories must be deleted, then all subdirectories (now empty) of the directory must be deleted, and then the directory (now empty) itself may be deleted.

Continuing with the example of the 'del dir' file system command, the decomposition module 303 finds corresponding log constraints, 'Predecessor-

Successor' and 'Parcel', in the Constraints list 314. In this particular example, 'Predecessor-Successor' indicates that the order of execution of the primitive actions must be maintained. The 'Parcel' log constraint indicates that the primitive actions must be executed as a group; i.e., if a file cannot be deleted for some reason, none of the other primitive actions may be executed. Another type of log constraint not shown in the look-up table 308 is an 'Alternatives' log constraint, which indicates that one action in the set of primitive actions may be chosen. Other types of log constraints may be defined as may be suitable to a particular implementation.

The look-up table 308 may be generated and stored in system memory when the system is manufactured, or initialized at power-up. It will be appreciated by one skilled in the art that the look-up table 308 may be stored in memory as any type of data structure as may be known in the art. Some types of data structures are linked lists, doubly linked lists, arrays, and others. The data structure includes binary encoded data representative of the data in the look-up table 308, which is readable by a microprocessor. It is to be understood that the look-up table 308 is meant to illustrate a logical relationship among various data types, and does not imply any particular physical arrangement of the data in memory. Thus a data structure having the data shown in the look-up table 308 is not limited to any physical arrangement in memory.

Primitive actions and log constraints may be dynamic and/or be context sensitive. For example, a user or application program may specify a log constraint. In addition, the primitive actions may be selected or designed to take into account semantics a particular application program. For example, semantics associated with electronic mail (e-mail) files and directories may differ from the semantics of a file management system, and primitive actions may be implemented that account for such semantic differences. Thus, user constraints and application constraints may be applied to primitive actions. The recording module 305 stores user constraints and application constraints in the log 306 with corresponding primitive actions. Primitive actions may change depending on the context of the file system. For example, primitive actions related to a 'rename' command may differ depending on whether the object being renamed is a directory or a file, the nature of the directory or file, and whether a file or directory already exists under the new name. Other systems and methods may be used besides, or in addition to, the look-up table 308 to translate file system commands into primitive actions and log constraints. For example, in a 'C' code implementation, the file system command may be 'switched' on to choose a set of primitive actions. After the RFS 302 determines the primitive actions, the recording module 305 may create, format, or encode the primitive actions into a form, such as Java objects, which is readable and/or executable by a reconciliation engine for reconciling the primitive actions with other primitive actions.

In one implementation, the log 306 includes a list of primitive actions and a list of log constraints. In another implementation, the log 306 is a series of Java objects wherein the primitive actions are embodied in executable code. As discussed above, the log 306 is readable by a reconciliation engine (e.g., the reconciliation engine 316).

In an exemplary implementation, file system commands from the I/O module 304 may be parceled together. A parcel refers to two or more file system commands or actions that are bundled together. When two or more commands or actions are parceled, they are to be executed together. File system commands may be parceled by a user or application in situations where the operations of the file system commands should not be separated. For example, a user may parcel a 'write file' command with a 'del file' command. In a situation such as this, where two file system commands are parceled, the corresponding primitive actions will be parceled by the RFS 302. Thus, the RFS 302 will create a 'parcel' log constraint to parcel the primitive actions corresponding to the parceled file system commands.

Fig. 4 illustrates an exemplary reconciliation engine 402 receiving a first exemplary log 404 and a second exemplary log 406, and generating a non-conflicting schedule 408 of primitive actions. As shown in Fig. 4, the first log 404 relates to a desktop computer, and the second log 406 relates to a personal digital assistant (PDA). The reconciliation engine 402 analyzes the desktop computer log

404 and the PDA log 406, determines if a conflict is created by the logs 404, 406, and resolves any conflicts that may arise. In the particular situation exemplified in Fig. 4, it will be seen that the desktop computer log 404 and the PDA log 406 do not create a conflict.

As shown, primitive actions in the desktop computer log 404 indicate creation of a directory named 'bar', creation of a file named 'foo', and a log constraint indicates that the two primitive actions have a 'Predecessor-Successor' relationship; i.e., directory 'bar' must be created before the file 'foo' is created to go in the directory. The PDA log 406 includes primitive actions to create a directory named 'foo2', create a file named 'bar2', and a log constraint indicating that the two creation actions have a 'Predecessor-Successor' relationship.

The reconciliation engine 402 uses object constraints (discussed below) to determine whether the primitive actions of the desktop computer log 404 conflict with those of the PDA log 406, and vice versa. In this particular example, the file system object constraints do not indicate a conflict because neither the files nor the directories share common names at common directory levels. Thus, the reconciliation engine 402 generates the non-conflicting schedule 408, which includes all the primitive actions from both the desktop computer log 404 and the PDA log 406.

Fig. 5 illustrates another exemplary reconciliation engine 502 receiving a first exemplary action log 504 and a second exemplary action log 506, and

proposing a first exemplary schedule 508 of non-conflicting primitive actions and a second exemplary schedule 510 of non-conflicting primitive actions. As in Fig. 4, a desktop computer action log 504, and a PDA action log 506 are illustrated. In the particular example shown in Fig. 5, and in contrast to the example in Fig. 4, the desktop computer action log 504 and the PDA action log 506 create a file system state conflict.

As shown in Fig. 5, the desktop computer log 504 includes primitive actions to create a directory named 'bar', create a file named 'foo', a log constraint indicating a 'Predecessor-Successor' relationship, and a log constraint indicating a 'Parcel' relationship. Also shown is the PDA log 508 having a primitive action to create a file named 'bar.'

In the example illustrated in Fig. 5, the reconciliation engine 502 uses object constraints to analyze primitive actions from the two logs 504 and 506. The object constraints may originate from any of a number of sources. In one implementation, the reconciliation engine 502 receives object constraints from a reconcilable file system (RFS) (e.g., RFS 216 or RFS 222, Fig. 2), an RFS proxy (e.g., RFS proxy 213, Fig. 3), or a reconciliation interface (e.g., reconciliation interface 220 or reconciliation interface 226, Fig. 2). As discussed herein, the object constraints indicate relationships between file system actions or commands from separate client devices, and the object constraints may be based on semantics associated with particular applications.

For example, the reconciliation engine 502 could receive an object constraint of 'mutually exclusive' with regard to names of directories and files at a common directory level. In other words, the object constraint indicates that a directory and a file cannot share the same name in the same directory. As a result, in this case, the reconciliation engine 502 identifies a conflict between the primitive action 'create dir bar' in the desktop computer log 504 and the primitive action 'create file bar' in the PDA log 506.

In this example, because the object constraint indicated a 'mutually exclusive' dependency, the reconciliation engine 502 identifies two possible non-conflicting schedules of primitive actions. Schedule A 508 includes the primitive actions and log constraint from the desktop computer log 504. Schedule B 510 includes the primitive action from the PDA log 506. The reconciliation engine 502 proposes schedule A 508 and schedule B 510 to the user to allow the user to select one of the schedules. In a particular implementation of the reconciliation engine 502, a user interface is provided, which enables a user to view proposed schedules, select one of the proposed schedules, and/or edit proposed schedules. The process of selecting one of the schedules can also be executed automatically using rules provided by the application programmer or the users.

In addition, the user is allowed to selectively 'undo' (i.e., roll back) any of the primitive actions. For example, the user may realize the creation of the file 'bar' is a mistake, and undo the 'create file bar' from the PDA log 504. By

undoing previously entered file system commands, conflicts between primitive actions may be resolved. Because the 'Parcel' log constraint indicates that the two actions in the desktop computer log 506 cannot be separately executed, if the user selectively undoes the 'create dir bar' action from the desktop computer log 506, the primitive action 'create file foo' will also be undone because of the 'Parcel' log constraint. An exemplary user interface that enables a user to edit a schedule of actions, including undoing unwanted actions, is discussed in further detail with reference to Figs. 10 and 11.

After the user selects one of the proposed schedules and/or undoes conflicting primitive actions, the resulting schedule of non-conflicting actions will be transmitted to the desktop computer and the PDA that generated the logs 506, 504. The desktop computer and the PDA will commit the resulting schedule, thereby creating identical file systems in the memories of the desktop computer and the PDA, respectively.

Exemplary Operations and User Interfaces for Reconciling File System States and/or Undoing File System Commands

Fig. 6 illustrates an exemplary reconciliation operation 600 for reconciling two file system states using primitive actions, log constraints, and object constraints. In general, primitive actions are logged at two or more sites as discussed herein, and are later reconciled to resolve any conflicts that may arise.

After a start operation 602, a log operation 604 logs primitive actions. Logging primitive actions may involve receiving a file system command and decomposing the file system command into corresponding primitive actions. The log operation 604 may also involve recording and/or transmitting the primitive actions to a reconciliation engine. A particular embodiment of the log operation 604 are illustrated in Fig. 7 and discussed in more detail below.

A reconcile operation 606 receives the logged primitive actions from two or more computing devices and reconciles the primitive actions. The reconcile operation 606 may involve collecting object constraints, identifying conflicts between two primitive actions, and resolving the identified conflicts. The reconcile operation 606 may also involve generating one or more non-conflicting schedules of primitive actions and proposing the schedules to a user. Particular embodiments of the reconcile operation 606 are shown and described in Figs. 8, 9, and 10. The reconciliation operation 600 ends at an end operation 608.

Fig. 7 illustrates an exemplary log operation 700 for generating one or more primitive actions representing a file system command. In general, the log operation 700 translates a file system command into primitive actions, stores the primitive actions, and sets log constraints corresponding to the primitive actions. It is assumed that a file system command has been received. The file system command, or a pointer, or other reference to the file system command is input to the log operation 700.

After a start operation 702, a translate operation 704 generates one or more primitive actions based on a file system command. In one embodiment, the translate operation 704 accesses a look-up table (e.g., the look-up table 308, Fig. 3) to identify primitive actions corresponding to the file system command. In another embodiment of the translate operation 704, a function call is made to an object representing the file system command, and the function returns a set of corresponding primitive actions. The primitive actions that are generated in the translate operation may be objects, such as Java objects, having methods and data for carrying out the primitive actions.

A store operation 706 stores the generated primitive actions in memory. In one embodiment, the primitive actions are recorded in a file. In this embodiment, the actions are embodied as primitive action identifiers that may be stored as encoded data in a file. Thus, the store operation 706 may create and/or open a file and append the primitive actions into the file. A primitive action is typically encoded in a file with an action identity, an action type or object method performed, any associated target object(s), and any associated arguments. This file is one exemplary embodiment of a log, such as the log 118 in Fig. 1, or the log 506 in Fig. 5.

In another embodiment of the store operation 706, the primitive actions are embodied as Java objects. In this embodiment, the store operation 706 stores references or pointers to the primitive action Java objects in a log for storing

tentative updates to the file system state. In object form, an action contains an action identity, a reference to the code of the method performed, references to the target object(s), and the action's arguments in either literal or pointer form.

A set operation 708 sets log constraints in a log to relate the primitive actions. Exemplary log constraints are illustrated in the look-up table 308 in Fig. 3. The set operation 708 may determine the log constraints based on a table, such as the look-up table 308, or by some other method. Log constraints may originate from a user or application program. Thus, the set operation 708 may store user constraints, application constraints, and/or other log constraints in a log, along with the associated primitive actions.

Fig. 8 illustrates an exemplary reconciliation operation 800 for resolving file system state conflicts that may exist among file system states of multiple computing devices. In general, the reconciliation operation 800 collects object constraints, generates a non-conflicting schedule, and commits the non-conflicting schedule.

After a start operation 802, a collect operation 804 collects object constraints that relate types of primitive actions from different computing devices. In one embodiment of the collect operation 804, a reconciliation engine (e.g., the reconciliation engine 112, Fig. 1, the reconciliation engine 402, Fig. 4, or the reconciliation engine 502, Fig. 5) calls a reconcilable file system (RFS) (e.g., the RFS 104, or the RFS 108, Fig. 1, or the RFS 302, Fig. 3) with a request for object

constraints that apply to two primitive actions. In another embodiment, the reconciliation engine may pass a 'del dir' primitive action and a 'create dir' primitive action to the RFS, and ask the RFS whether the two actions can be executed simultaneously. In this embodiment of the collect operation 804, the RFS may generate object constraints that dictate whether two primitive actions conflict.

Exemplary object constraints are illustrated in Table 1, Table 2, and Table 3. The tables show dependencies on files and directories for different types of actions. In Table 1, the types of actions are grouped into actions that link two file system nodes and actions that unlink two file system nodes. For example, the heading "Link/Link" refers to two actions that each link a directory or a file into a parent directory in a file system tree. As another example, the heading "Link/Unlink" refers to an action that links a file or a directory into a parent directory, and an action that unlinks a file or directory from a parent directory.

Table 1 shows object constraints that apply to two actions, in which both actions link or unlink a child file or child directory into a parent directory. As shown in the Table 1, when the two children have different names and/or different parent directories, the entry "No Relation" is used to indicate absence of any constraint between the two actions associated with the children. Also shown in Table 1, if the two children have the same parent and the same name, the object constraint depends on the action type.

When both children are to be linked to the same node under identical names, as shown under the heading "Link/Link", the object constraint is "Mutually Exclusive," which means that linking one of the children under some name necessarily precludes linking the other child under the same name. When one child is to be linked and the other unlinked from a same parent directory, the object constraint is "Best Order," meaning that a preferred order of linking and unlinking exists and is determined by the RFS. In one particular implementation, the "Best Order" involves applying unlinks before links. When both affected children are to be unlinked from their parent directories, as shown under the heading "Unlink/Unlink," the object constraint is "Commute," meaning that the unlinking actions are commutative (i.e., may be executed in any order).

Table 1

Directory	Link/Link	Link/Unlink	Unlink/Unlink
Different parent, name	No Relation	No Relation	No Relation
Same parent & name	Mutually Exclusive	Best Order	Commute

The entries shown in Table 2 concern two actions that both affect a file, either the same file, or different files. The action types in Table 2 are grouped into actions that involve reading from a file and actions that involve writing to a file. As indicated by the row labeled "Other File" in Table 2, if the two actions affect

different files, the object constraint is "No Relation", regardless of the type of action (i.e., Read or Write). In the row labeled "Same File," the object constraint depends on the type of action. When the primitive action involves reading the same file, the object constraint is "Commute."

When one primitive action involves a read from the file and the other primitive action involves a write to the file (i.e., the "Read/Write" column), the object constraint is "Best Order," wherein the RFS can choose a preferred order. In a particular RFS implementation, the "Best Order" is read actions before write actions. If the two actions involve writes to the file, as shown under the heading "Write/Write," the object constraint is "Mutually Exclusive," meaning that there is a conflict and that only one write action is allowed to be executed.

Table 2

File	Read/Read	Read/Write	Write/Write
Other File	No Relation	No Relation	No Relation
Same File	Commute	Best Order	Mutually Exclusive

The entries shown in Table 3 concern an action that affects a file, and an action that affects a directory. The action types in Table 3 are grouped into actions that involve unlinking a directory and writing to a file, and other actions. As indicated by the row labeled "Other File" in Table 3, if the two actions affect a file

that is not in the affected directory, the object constraint is "No Relation", meaning that the unlinking action does not affect the writing action, and vice versa. In the column labeled "Other Actions," which covers linking actions, reading actions, and others, the object constraint is "No Relation," regardless of whether the affected file is in the affected directory. As shown in the column with heading "Unlink/Write," if the affected file is in the affected directory, and action to unlink the directory from the node has a "Mutually Exclusive" relation to the action to write to the file, meaning that a conflict exists between the two actions.

Table 3

Directory/File	Unlink/Write	Other Actions
Other File	No Relation	No Relation
Same File	Mutually Exclusive	No Relation

The exemplary object constraints shown in Table 1, Table 2, and Table 3 represent directory and file relationships for a particular RFS implementation. Other RFS implementations may have different object constraints than those shown above. Further, other RFS implementations may define object constraints for pairs of file system objects, besides, or in addition to, directories and files. For example, in a particular RFS implementation, it may be meaningful to define object constraints that relate primitive actions affecting file types that contain e-

mail messages, file types that contain calendar entries, and so on. In addition, Table 1, Table 2, and Table 3 may be embodied using software, hardware, firmware, or any combination thereof. In a particular embodiment, the tables are stored as an array of entries in random access memory (RAM). Any data structure as may be known in the art may be used to implement the data shown in the above tables.

In one embodiment of the collect operation 804, the RFS receives a request, for example from the reconciliation engine, to determine whether a conflict exists between two actions. The RFS determines the actions types (e.g., Read, Write, Link, Unlink, etc.), and the file object types (e.g., file, directory, etc.), and accesses the tables above to find the appropriate object constraint. The RFS then responds to the request by sending the object type back to the reconciliation engine.

A generate operation 806 generates one or more non-conflicting schedules using the primitive actions, the log constraints, and the object constraints. In one embodiment, if the object constraint indicates that two actions are mutually exclusive, the generate operation 806 may choose one of the actions to insert in the schedule. In this embodiment, the choice of which of two actions to insert may be based on a value assessment of the two actions. For example, an action that deletes a directory may be less valuable than an action that writes to a file in the directory; thus, the writing action would be selected to insert into the schedule.

Exemplary value assessment algorithms are discussed in further detail with respect to Fig. 9.

In another embodiment of the generate operation 806, the user is prompted with a proposed schedule of primitive actions. The user may accept or reject the proposed schedule. If the user rejects the proposed schedule, the generate operation 806 generates another proposed schedule, and so on, until the user accepts a proposed schedule. The user may also selectively undo actions in the proposed schedule. Exemplary embodiments of the generate operation 806 are shown and discussed in further detail in Fig. 9.

A commit operation 808 commits the non-conflicting schedule of primitive actions from the generate operation 806. The commit operation 808 involves communicating the non-conflicting schedule to computing devices (e.g., the PDA 102, Fig. 1, or the desktop computer 106, Fig. 1) and executing the scheduled actions on the computing devices.

In one embodiment of the commit operation 808, the non-conflicting schedule of primitive actions that were generated in the generate operation 806 is executed from the previously committed state by a reconcilable file system (e.g., the RFS 104, or the RFS 108, Fig. 1, or the RFS 302, Fig. 3). In this embodiment, any tentative updates that were made to the previously committed file system state may be repealed to arrive at the previously committed state, prior to executing the non-conflicting schedule of primitive actions.

To repeal actions in the commit operation 808, an RFS access a log (e.g., the log 118, Fig. 1, the log 120, Fig. 1, or the log 306, Fig. 3) to identify any tentative updates, and reverse the tentative updates to arrive at the previously committed state before re-playing the primitive actions in the non-conflicting schedule.

Thus, after multiple computing devices execute the non-conflicting schedule starting from the same previously committed file system state, the multiple computing devices have a new common committed file system state. The commit operation 808 may be viewed as a synchronizing operation, wherein the file system states of multiple computers are made to match or harmonize. The commit operation 808 may also be viewed as a replicating operation, wherein a file system replica is created on one or more computing devices.

Fig. 9 illustrates an exemplary reconciliation operation 900 for reconciling one or more logs of primitive actions. Generally, the reconciliation operation 900 generates a schedule of primitive actions based on an input set of primitive actions. More specifically, the exemplary reconciliation operation 900 iterates through each of the primitive actions in the input set, adding a selected input action to a schedule if the input action has the highest value among all the input actions, and if the selected input action can be successfully executed.

After a start operation 902, a receiving operation 904 receives a list of candidate primitive actions. The list of candidate primitive actions includes

actions representing changes to file system states on one or more devices (e.g., the PDA 102 or the desktop computer 104, Fig. 1). As discussed above, the primitive actions are recorded in action logs on the devices. When more than one action log is being reconciled, the list of candidate actions received in the receiving operation 904 is a union of the plurality of action logs. The receiving operation 904 saves the list of candidate actions in memory, so that the list can be accessed and/or updated during the reconciliation operation 900. The receiving operation also stores the set of known constraints (log constraints and object constraints) that apply to the candidate actions.

An obtaining operation 906 obtains a checkpoint of the current file system state. The checkpoint of the file system state is the state at which the reconciliation operation 900 will begin analyzing and reconciling the primitive actions in the list of candidate actions. Typically the checkpoint is a previously committed file system state. The checkpoint state may be stored in memory and updated during the reconciliation operation 900 as candidate actions are executed

A selecting operation 908 selects one of the candidate actions from the candidate action list based on the value assessment of the candidate actions, and the log and object constraints. Generally, an action is selected that does not violate the log constraints or object constraints of previously executed actions. A designer of the RFS system may specify any number of value criteria suitable to the particular implementation. These criteria may be specified in schemata,

programmatically, or other declarative means. During operation, the selecting operation applies the specified criteria to the candidate actions to identify the action with the highest value.

In the selecting operation 908, values are generated for each of the candidate actions, and the action with the highest associated value is selected. The selecting operation 908 may involve attributing a higher value to a first candidate action than a second candidate action if scheduling the first candidate action would result in fewer conflicts with scheduled actions than would scheduling the second candidate action. The log and object constraints are analyzed to determine the number of conflicts that arise for each candidate action. For example, a tally of conflicts can be generated for each action as indicated by the action's associated log and object constraints; the action with the lowest tally has the highest value.

In one implementation of the selecting operation 908, an action's value is based on the value of actions that are schedulable before and/or after the action, the value of alternatives to the action, and/or the value of actions that are mutually exclusive to the action. Thus, for example, the value of an action 'a' may be determined to be higher than another action 'b' if:

1. The value of actions that can only be scheduled before 'a' is lower than the value of those actions that can only be scheduled before 'b',

2. The value of alternatives to 'a' is lower than the value of alternatives to 'b,'

3. The value of actions mutually exclusive with 'a' is lower than the value of actions mutually exclusive with 'b,' or

4. The value of actions that can only be scheduled after 'a' is higher than the value of actions that can only be scheduled after 'b.'

An executing operation 910 receives the selected candidate action from the selecting operation 908 and executes the selected candidate action. The executing operation 910 executes the selected candidate action from the checkpoint file system state. Execution of the selected candidate action may cause the file system state to change. Thus, the executing operation 910 updates the file system state to the new checkpoint file system state. The executing operation 910 generates results, such as 'success' or 'failure', which indicate the outcome of the execution operation 910.

An execution results query 912 receives the execution results from the executing operation 910 and determines whether the execution of the selected candidate action succeeded. If the execution results query 912 determines that the

execution of the selected candidate action succeeded, the reconciliation operation 900 branches "YES" to an adding operation 914.

The adding operation 914 appends the selected candidate action to a schedule of actions. If no candidate actions had been scheduled prior to the adding operation 914, the adding operation 914 creates a schedule (e.g., a file) and inserts the selected candidate action as the first entry. If actions have already been scheduled, the selected candidate action is appended after the last scheduled action.

A removing candidate action operation 916 removes the selected candidate action from the list of candidate actions so that the selected candidate action is not analyzed again. A removing conflicting actions operation 918 removes actions from the list of candidate actions that conflict with (i.e., violate constraints against) the selected candidate action, to avoid later attempting to reconcile the conflicting actions with scheduled actions. For example, if the selected candidate action creates a file in a directory, another candidate action that deletes the directory would conflict with the selected candidate action, and would be removed from the candidate list. By removing the conflicting actions from the list of candidate actions, time is saved during the reconciliation operation 900, and the user may be assured that the final proposed schedule contains only non-conflicting actions.

A removing precondition actions operation 920 removes any actions from the list of candidate actions that are not permitted to run after the selected

candidate action. Any action that is not permitted to run after the selected action, and that has not been scheduled already cannot be scheduled after the selected action without violating an ordering constraint. Such a situation arises when an action has an associated Predecessor-Successor constraint. For instance, if a create file action has a create directory action as a predecessor, the create directory action is removed.

A last candidate query 922 determines whether the selected action is the last candidate action in the list of candidate actions. If the selected action is the last action in the list of candidate actions, the reconciliation operation 900 branches "YES" to an ending operation 924 where the reconciliation operation 900 ends. If the selected action is not the last action in the list of candidate actions, the reconciliation operation 900 branches "NO" to the selecting operation 908, wherein a next candidate action in selected from among the list of candidate actions. As previously described the selecting operation 908 selects that candidate action that has the highest value.

Referring again to the execution results query 912, if it is determined that the execution of the selected candidate action did not succeed, the reconciliation operation 900 branches "NO" to a side effects query 926. The side effects query 926 determines whether any side effects resulted from the attempted execution of the selected candidate action.

If the side effects query 926 determines that side effects did occur during execution of the selected candidate action, or of some action that is in a same parcel as the candidate action, the reconciliation operation 900 branches "YES" to a rolling back operation 928. The rolling back operation 928 reverses or repeals the side effects caused by the execution of the selected candidate action and any actions in a parcel with the candidate action.

One embodiment of the rolling back operation 928 iteratively executes a single action at a time to reverse any side effects. If a single action cannot be executed to eliminate a side effect, then the rolling back operation 928 returns to the last saved checkpoint and restores any intervening actions to the candidate list.

If, in the rolling back operation 928, the selected action is part of a parcel, and actions of this parcel have been executed in the same schedule, then those actions too must be removed from the schedule and their side-effects (if any) rolled back. The reconciliation engine can roll back a parcel of actions by iterating over each executed action in the parcel and undoing the action's side effects, in reverse chronological order. Alternatively, the reconciliation engine can return to the last checkpoint and resume scheduling from there.

A removing selected action operation 930 removes the selected action from the list of candidate actions. If the side effects query 926 determines that side effects did not occur during execution, the reconciliation operation 900 branches "NO" to the removing selected action operation 930.

During or after the reconciliation operation 900, the user may be prompted to selectively edit actions in a proposed action schedule. A user interface is presented to the user, through which the user can view a list of candidate actions, scheduled actions, and constraints, and selectively edit the actions and constraints. Through the interface, the user may undo one or more actions related to commands that were mistakenly entered. The user may undo undesirable actions that conflict with other desirable actions, so that the desirable actions remain in the schedule. An exemplary 'undo' user interface is illustrated in Fig. 10.

Fig. 10 illustrates an exemplary graphical user interface (GUI) 1000 for enabling a user to selectively edit primitive actions. The GUI 1000 is an interface to a reconciliation engine, such as the reconciliation engine 112 in Fig. 1, or the reconciliation engine 212 in Fig. 2.

The exemplary GUI 1000 has three windows: a state window 1002, a proposed actions window 1004, and a constraints window 1006. The state window 1002 lists the file system state of one or more file systems. The proposed actions window 1004 lists one or more primitive actions that resulted in the state(s) in the state window 1004. The constraints window 1006 lists constraints associate with two of the actions in the proposed actions window 1004.

The user can guide the pointer 1008 over actions in the proposed actions window 1004 and select one or more of the actions. The user may undo one or more of the actions using an undo menu selection. If the user undoes an action, the

action and any dependent actions are removed from the proposed actions. A dependent action is an action that depends on another action through some constraint. For example, as shown in the GUI 1000, the two actions, "Recreate file," and "Link file" are in a parcel as indicated by the "SuccessorParcel" constraint in the constraint window 1006. Thus, if the user undoes either of the actions in the actions window 1004, the other actions will also be deleted, because they are to execute as a parcel, if they execute at all.

Fig. 11 illustrates reconciliation/undo window 1100 through which a user can selectively undo unwanted commands. The GUI 1100 may be viewed as a global undo interface useful to take into account possible dependencies across documents. The GUI 1100 includes a number of actions displayed as selectable action icons 1102 and user input mechanisms presented by a reconciliation engine. Branches in this tree placed vertically with respect to one another represent independent actions. Vertically-placed actions can run in any order with respect to one another. Horizontal node placement represents actions that are related by a causal dependency, running left to right. Horizontally-placed actions must be executed in the prescribed order, left first then rightwards.

The reconciliation engine presents the actions after a scheduling operation, such as is shown above. The reconciliation engine presents the action icons 1102 in a hierarchy. At a root node in the hierarchy is a first make directory action icon 1104. The first directory at the root node is labeled "Reconciliation Engine."

A second make directory action 1106 branches off the root node to create a "documents" directory under the root "Reconciliation Engine" directory. The GUI has placed the make directory action 1106 to the right of the make directory action 1104, indicating that the latter must be executed before the former. A third make directory action icon 1108 branches off the root node to create an "images" directory under the root "Reconciliation Engine" directory. The GUI has placed action 1108 vertically underneath action 1106, indicating that they are independent and can execute in any mutual order. Under the "documents" directory, a calendar is created, as represented by a "create calendar" action 1110. A fourth make directory action 1112 creates a "Word" directory under the "documents" directory.

Under the "Word" directory, a document is created by a create file action 1114. After the document is created, a write action 1116 writes to the document. The reconciliation engine decomposes the write action 1116 into a break out window 1118, including a number of actions that make up the write action 1116. The write action 1116 initially includes a new document action 1120. The new document action 1120 opens a blank document for editing.

After the blank document is opened, there are two branches of editing. The top branch includes a first typing action 1122, in which paragraph 1 is added to the blank document. Next, a bolding action 1124 adds bolding to the first paragraph. In a bottom branch off the new document action 1120, a second typing action 1126 adds a second paragraph to the blank document. The tree structure indicates that

the two paragraphs are independent from each other. A user may select any of the action icons 1102-1126, and undo (i.e., remove) them from the schedule of actions.

Using an input mechanism, such as an undo button 1128, the user may undo an action corresponding to a selected action. When the user depresses the undo button 1128, the reconciliation engine creates a new schedule without the selected actions and without their dependents. Creating a new schedule can be done as shown and described in Fig. 9. During the process of creating the new schedule, any actions that depend upon the action that is selected to be undone will not be included in the new schedule. The reconciliation presents the new schedule to the user in a hierarchical fashion.

The user may then "redo" the previously undone actions by selecting a redo button 1130. In response to receiving a redo input from the user, the reconciliation recreates the previous schedule, this time including the previously undone actions, unless they are excluded for other reasons. For instance, a conflict may have appeared in the meantime, which prevents one or more of the previously undone actions from being redone.

An Exemplary Operating Environment

Fig. 12 and the corresponding discussion are intended to provide a general description of a suitable computing environment in which the described arrangements and procedures to reconcile and/or undo file system states may be

implemented. Exemplary computing environment 1220 is only one example of a suitable computing environment and is not intended to suggest any limitation as to the scope of use or functionality of the described subject matter. Neither should the computing environment 1220 be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in the exemplary computing environment 1220.

The exemplary arrangements and procedures to reconcile file systems between interconnected components are operational with numerous other general purpose or special purpose computing system environments or configurations. Examples of well known computing systems, environments, and/or configurations that may be suitable for use with the described subject matter include, but are not limited to, personal computers, server computers, thin clients, thick clients, hand-held or laptop devices, multiprocessor systems, microprocessor-based systems, mainframe computers, distributed computing environments such as server farms and corporate intranets, and the like, that include any of the above systems or devices.

The computing environment 1220 includes a general-purpose computing device in the form of a computer 1230. The computer 1230 may include and/or serve as an exemplary implementation of a reconciliation engine and/or a reconcilable and undoable file system described above with reference to Figs. 1-11. The components of the computer 1230 may include, by are not limited to, one

or more processors or processing units 1232, a system memory 1234, and a bus 1236 that couples various system components including the system memory 1234 to the processor 1232.

The bus 1236 represents one or more of any of several types of bus structures, including a memory bus or memory controller, a peripheral bus, an accelerated graphics port, and a processor or local bus using any of a variety of bus architectures. By way of example, and not limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnects (PCI) bus also known as Mezzanine bus.

The computer 1230 typically includes a variety of computer readable media. Such media may be any available media that is accessible by the computer 1230, and it includes both volatile and non-volatile media, removable and non-removable media.

The system memory includes computer readable media in the form of volatile memory, such as random access memory (RAM) 1240, and/or non-volatile memory, such as read only memory (ROM) 1238. A basic input/output system (BIOS) 1242, containing the basic routines that help to communicate information between elements within the computer 1230, such as during start-up, is stored in ROM 1238. The RAM 1240 typically contains data and/or program modules that

are immediately accessible to and/or presently be operated on by the processor 1232.

The computer 1230 may further include other removable/non-removable, volatile/non-volatile computer storage media. By way of example only, Fig. 12 illustrates a hard disk drive 1244 for reading from and writing to a non-removable, non-volatile magnetic media (not shown and typically called a "hard drive"), a magnetic disk drive 1246 for reading from and writing to a removable, non-volatile magnetic disk 1248 (e.g., a "floppy disk"), and an optical disk drive 1250 for reading from or writing to a removable, non-volatile optical disk 1252 such as a CD-ROM, DVD-ROM or other optical media. The hard disk drive 1244, magnetic disk drive 1246, and optical disk drive 1250 are each connected to bus 1236 by one or more interfaces 1254.

The drives and their associated computer-readable media provide nonvolatile storage of computer readable instructions, data structures, program modules, and other data for the computer 1230. Although the exemplary environment described herein employs a hard disk, a removable magnetic disk 1248 and a removable optical disk 1252, it should be appreciated by those skilled in the art that other types of computer readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, random access memories (RAMs), read only memories (ROM), and the like, may also be used in the exemplary operating environment.

A number of program modules may be stored on the hard disk, magnetic disk 1248, optical disk 1252, ROM 1238, or RAM 540, including, by way of example, and not limitation, an operating system 1258, one or more application programs 1260, other program modules 1262, and program data 1264. Application programs 1260 may include file system management software, such as a reconcilable file system and/or a reconciliation engine for managing file system objects and reconciling file system commands with other file system commands, as discussed herein.

A user may enter commands and information into the computer 1230 through optional input devices such as a keyboard 1266 and a pointing device 1268 (such as a "mouse"). Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, serial port, scanner, or the like. These and other input devices are connected to the processing unit 1232 through a user input interface 1270 that is coupled to the bus 1236, but may be connected by other interface and bus structures, such as a parallel port, game port, or a universal serial bus (USB).

An optional monitor 1272 or other type of display device is connected to the bus 1236 via an interface, such as a video adapter 1274. In addition to the monitor, personal computers typically include other peripheral output devices (not shown), such as speakers and printers, which may be connected through output peripheral interface 1275.

The computer 1230 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 1282. The remote computer 1282 may include many or all of the elements and features described herein relative to the computer 1230. The logical connections shown in Fig. 12 are a local area network (LAN) 1277 and a general wide area network (WAN) 1279. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets, and the Internet.

When used in a LAN networking environment, the computer 1230 is connected to the LAN 1277 via a network interface or an adapter 1286. When used in a WAN networking environment, the computer 1230 typically includes a modem 1278 or other means for establishing communications over the WAN 1279. The modem 1278, which may be internal or external, may be connected to the system bus 1236 via the user input interface 1270 or other appropriate mechanism. Depicted in Fig. 12 is a specific implementation of a WAN via the Internet. The computer 1230 typically includes a modem 1278 or other means for establishing communications over the Internet 1280. The modem 1278 is connected to the bus 1236 via the interface 1270.

In a networked environment, program modules depicted relative to the personal computer 1230, or portions thereof, may be stored in a remote memory storage device. By way of example, and not limitation, Fig. 12 illustrates remote application programs 1289 as residing on a memory device of remote computer

1282. It will be appreciated that the network connections shown and described are exemplary and other means of establishing a communications link between the computers may be used.

Conclusion

Although the described arrangements and procedures to reconcile file systems interconnected components have been described in language specific to structural features and/or methodological operations, it is to be understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or operations described. Rather, the specific features and operations are disclosed as preferred forms of implementing the claimed present subject matter.